Targeted expressions: safe object creation with void safety

Bertrand Meyer[1]

Draft 2.3 (30 July 2012). This is a working document, not a polished article.

1 Introduction

In a language supporting void safety [1], object creation poses a particular problem, since a creation procedure (constructor) may be cause calls to non-fully-initialized objects and hence cause crashes (several examples follow). We may call this problem the fragile delegate dilemma by analogy with [2].

A solution to this problem for Java was proposed by Müller [3, 4], but it involves a significant language extension (**commit** and **free** keywords) and it seems unrealistic that programmers would go for it. We have developed a solution that does not rely on any new syntax or keywords; it is simply an extra validity rule. Existing harmless code will pass the rule; code that could cause improper object access will trigger a violation, which it should be possible to phrase in such a way that ordinary programmers will see what the problem is and will be able to correct it easily.

Section 2 gives a simple example. Section 3 presents the rules. Section 4 lists all the examples of which we are aware and shows that the rule handles them correctly. Section 5 lists some issues to be discussed.

2 A simple example

The following schemes for the EiffelVision library [5] are unsound:

In *EV_ANY*:

```
implementation: EV_ANY_IMP


default_create              -- Version 1; warning: invalid.
        do
                create_interface_objects
                create implementation.make (Current)                -- [S1]
                initialize
        end
```

[1] Work done with Alexander Kogtenkov and Emmanuel Stapf. I am responsible for the writing of the current text.

```
default_create                -- Version 2; warning: invalid

        do
                create_interface_objects

                create implementation                -- [S2]
                implementation.set_interface (Current)                -- [S3]
                initialize
        end
```

In *EV_ANY_IMP* (with *default_create* as creation procedure in version 2):

```
interface: EV_ANY          -- Warning: invalid in version 2

make (v: EV_ANY)           -- Warning: invalid
        do
                interface := v                                        -- [S4]
                interface.do_something                                -- [S5]
        end

set_interface (v: EV_ANY)
        do
                interface := v                                        -- [S6]
        end
```

The problems, causing invalidity that should be detected, are:

- In version 1, [S5] produces a qualified call on the *EV_ANY* object, known here through *interface*, before it is fully initialized.
- In version 2, this problem does not arise any more, but *interface* has to be declared as **detachable** *in EV_ANY_IMP* because it is not properly set at the end of the creation procedure *default_create* of *EV_ANY_IMP*, violating 8.9.17 of the Eiffel ISOstandard [6] (hereafter called ISO Eiffel). Then every subsequent use of *interface* needs to be protected by an object test, including after initialization even though the problem will not arise then.

An appropriate rule must:

- Flag both version 1 and version 2 as invalid.
- Accept version 1 if [S5] is removed. It is OK to set *interface* as part of creation, but only if no calls are made on *interface* until everything has been created. If a call such as [S5] is needed, it must (as in version 2) be performed subsequent to creation. Unlike in version 2, however, a mere assignment to a local attribute does not require such separation and can be performed as part of creation.

3 The rule

The new rule must be added to the existing validity rules of ISO Eiffel, and hence is not yet in the "if and only if" format used throughout that standard document. As elsewhere in ISO-EIffel, underlined terms correspond to concepts possessing a precise definition elsewhere in the document.

3.1 "Follows"

ISO-Eiffel defines the notion of an evaluation position "preceding" another. We say that *ep2* follows *ep1* if and only if *ep1* precedes *ep2*.

3.2 Definition: a feature is "*C*-***creation-involved***" for a class *C* if it is either a creation procedure of *C*, or is the feature of a Call in (recursively) a *C*-creation-involved feature, or is the redeclaration of (recursively) a C-creation-involved feature.

3.3. Definition: For a given class *C*, an expression appearing at an evaluation position *ep* in a *C*-creation-involved feature is **targeted** at *ep* if, after replacement of all expressions at *ep* by their equivalent dot forms, it is of an expanded type or appears, at an evaluation position *ep'* that follows *ep* or is in another *C*-creation-involved feature, in any of the following roles:

3.3.1. The target of an Object_call.
3.3.2. Either operand of an Equality expression using the operator **~** or /~.
3.3.4. The source of an assignment whose target is (recursively) targeted at *ep'*.
3.3.5. The target of an assignment whose source is (recursively) targeted at *ep'*, if there is no assignment with the same target at a position that follows *ep* and precedes *ep'*.
3.3.6. An actual argument of a Call or Precursor or Creation_call, or an Agent_actual of an agent, where the corresponding formal argument is (recursively) targeted at the first evaluation position of the feature of the call.
3.3.7 A formal argument of a routine *r*, if *ep* is the first position of *r* and some evaluation position *ep'* in a *C*-creation-involved feature contains a call to *r* such that the corresponding actual argument is (recursively) targeted at *ep'*.
3.3.8 If *ep* contains a Unqualified_call to a routine *r*, a variable that is (recursively) targeted at the end position of *r*.
3.3.9. The Expression of an Object_test, where the Object-Test local is (recursively) targeted at *ep'*.
3.3.10 The Expression of an Iteration (**across**...) where the Identifier is (recursively) targeted at *ep'*.
3.3.11 A subexpression of any expression that is (recursively) targeted at *ep'*.

3.4. **Rule**

If **Current** or an agent is targeted at an evaluation position of a creation procedure of *C*, then all attributes of *C* must be properly set.

3.6 **New feature**: in class *ANY*, we assume a boolean argument-less query *available*, which returns **True**. It is not clear whether we should make it frozen. Note that this facility is a convenience and plays no role in the rest of the discussion.

4 Examples

MP will refer to Müller's paper [3] and MS to the accompanying slides [4]. Since not all readers may have access to the slides, program examples from [4] will be copied here.

4.1 A simple case

The following simple example illustrates the application of the rule[2]:

```
class C create make feature

   a: A
   b: B              -- Set only at EP4. The details of class B do not matter.

   some_routine
                  do
                        create a.make (Current)    -- EP2
                        a.crash           -- EP3
                        create b          -- EP4
                  end
end

class A create make feature
   value: C
   make (c: C) do value := c end
   crash do print (value.b) end
end
```

This example is invalid (and would indeed cause a crash):

- crash is A-creation-involved (EP3)
- value is targeted in crash (3.3.1)
- Hence c in make is targeted at the beginning of make (3.3.4)
- Hence Current is targeted at EP2
- But at EP2 not all fields of the current object are properly set since EP4 has not been executed yet (b is not initialized), so rule 3.4.1 is violated.

[2] Note on the program examples in this draft: the Eiffel font conventions have only been applied in part. There are in particular inconsistencies in the use of italics and color, due mostly (as other other instances of bad formatting) to conversion from Google Docs to Microsoft Word.

4.2 Eiffelvision

In the EiffelVision example of section 1:

- interface is targeted at [5] (3.3.1)
- Hence it is targeted at [4] (definition of ep')
- Hence v is targeted at the beginning of make of EV_ANY_IMP (3.3.4)
- Hence in version 1 Current is targeted at [1]
- But the attributes are not all properly set (we assume that this is the purpose of initialize), so version 1 violates 3.4.1
- As we saw version 2 is also invalid, for reasons already specified in ISO Eiffel

4.3 MP, figure 1, page 5

No one would use ANY in Eiffel for the data, we would just write generic classes, but I have kept ANY for compatibility with MP.

```
class LIST create
        make
feature
            make
                do
                        create sentinel.make_empty (Current)
                end

            sentinel: NODE
            insert (data: ANY)
                        do sentinel.insert_after (data) end
end
```

In MP `data' is of type `detachable ANY' but I don't see the point and I have made it attached throughout.

```
class NODE create make, make_empty feature
            parent: LIST
            previous, next: NODE
            data: detachable ANY

  make_empty (p: LIST)
        do
            parent := p
            previous := Current
            next := Current
        end
```

```
        make (p, n: NODE, d: ANY)
                        -- There seems to be a missing precondition: p and n should
be chained to each other in
                        -- the parent list! However I am sticking to the original scheme.
                        -- I also don't see in the paper a class invariant stating that the
list is cyclic.
                        -- In fact I don't understand the role of the second argument n;
isn't it always expected
                        -- to be p.next, as in the example call in insert_after? If the
second argument is removed
                        -- there is no need for a precondition.
                        -- I assume this problem comes from the Leino-Fähndrich
paper.
                do
                        parent := p.parent
                        previous := p ; next := n ; data := d
                end

        insert_after (d: ANY)
                local
                        new: NODE
                do
                        create new.make (Current, next, d)
            next.set_previous (new)
            next := new
                end

   set_previous (p: NODE)
         -- Set `previous' to `p'.
      do
         previous := p
      end
end
```

This example does not violate any of the validity rules and hence is OK:

- In LIST, **Current** is not targeted since the argument p of make_empty in NODE is not targeted.
- **Current** is also not targeted in the creation procedures of NODE.

4.4 MP, figure 2, page 6

```
class C create make feature
        f, g: C

        set_f (q: C)   do f := q end

        make (p: C)
                do
                        set_f (p)              --              [EP1]
                        f.set_f (Current)  --                  [EP2]
                        g := p.f.g.f  -- Would trigger void call if permitted        [EP3]
                end
end
```

This class violates the rule:

- p is targeted at EP3 (3.3.1).
- Hence p is targeted at EP1 (definition of ep')
- Hence q is targeted in set_f because of the call at EP1 (3.3.4)
- Hence *f* is targeted at the end of *set_f* (3.3.5)
- Hence **Current** is targeted at EP2 (3.3.6)
- ·Attribute *g* is not properly set at EP2.
- ·As a consequence, rule 3.4.1 is violated.

4.5 MS, problem 1

I am not sure whether everyone has access to Müller's slides, so here is the original text

```
class Demo {
 Vector! cache;

 Demo( ) {
   int size = optimalSize( );
   cache = new Vector( size );
 }

 int optimalSize( ) {
   return 16;
 }
}
```

```
class Sub extends Demo {
 Vector! data;
 Sub( Vector! d ) {
   data = d.clone( );
 }

 int optimalSize( ) {
   return data.size( ) * 2;    // NULL POINTER EXCEPTION
 }
}
```

In Eiffel the direct translation is:

```
class DEMO create make feature

        cache: VECTOR

        make
                local
                        size: INTEGER
                do
                        size := optimal_size        -- [EP3]
                        create cache.make (size)
                                -- I don't see the need for the local variable size,
                                -- it just seems to complicate the code.
                end

        optimal_size: INTEGER do Result := 16 end

end

class SUB_DEMO inherit                          -- `Sub' in original
        DEMO
                redefine optimal_size end
create
        make
feature
        data: VECTOR

        make (d: VECTOR)                        -- Version A
                do
                        -- We will add instructions here, see below.
                  data := d.cloned              -- [EP4]
              end

        optimal_size: INTEGER
              do
                        Result := data.size * 2    -- [EP5]
```

```
                                        -- Would trigger void call in Java etc. if
            permitted
            end
end

class EXAMPLE feature
        test
                local
                        v: VECTOR
                        s: SUB_DEMO
                do
                        create v
                        create SUB_DEMO
                end
end
```

The problem comes in Java etc. from the strange property (inherited from C++) that *make* from *SUB_DEMO* automatically calls *make* from *DEMO*. This will not occur in EiffelTo get the same effect in Eiffel we have to make this call explicit (it's automatic in Java, a property coming from C++) by rewriting *make* of SUB_DEMO as follows:

```
        make    (d:   VECTOR)                   -- Version B, will cause trouble
                do
                   Precursor
                   data := d.cloned             -- [EP4]
            end
```

Then **Precursor** calls the new (dynamically bound) version of *optimal_size*, which needs to call *data.size*, where *data* has not yet been set, causing the void call. This is the same problem as occurs with the Java version.

As Müller points out the order of instructions in the Demo constructor in the Java version is not important (there could be some other scheme causing data to be accessed before it is properly initialized). The order of instructions in version B of SUB_DEMO's *make* in Eiffel is, however, relevant: if we reverse it, giving

```
        make    (d:   VECTOR)                   -- Version C, should be valid
                do
                   data := d.cloned             -- [EP4]
                   Precursor
            end
```

everything should be fine. The Java etc. version does not allow this change (in the way it is written) since the automatic call of the inherited constructor occurs first by construction.

The new rules are not needed in this example. In version B, *data* is not properly set (ECMA-367 8.19.17) when used in the redefined *optimal_size*. The problem does not arise with Version C.

4.6 MS, problem 2 (observer pattern)

Again here is the original text from the slides.

```
class Demo implements Observer {
 static Subject! subject;
 Demo( ) {
   subject.register( this );
 }
 void update( … ) { }
}

class NewDemo extends Demo {
 Vector! data;
 NewDemo( Vector! d ) { data = d.clone( ); } // starts with implicit parent constructor call,
causes null pointer exception (BM comment)
 void update( … ) { … data.size( ) … }
}
```

Note: For class NewDemo the name in Müller's slides is Sub but I have changed the name since Sub is confusing (it evokes `Subject').

```
class Subject {
 void register( Observer! o ) {
    …
    o.update( … );
 }
}
```

Here is the Eiffel version.

```
class DEMO inherit
        OBSERVER
create
        make
feature

        subject: SUBJECT

        make
            do
                subject.register (Current)          -- [EP5]
            end

        update (...) do end
end
```

```
class NEW_DEMO inherit
        DEMO
                redefine make, update end
creation
        make
feature
        data: VECTOR

        make (d: VECTOR)
            do
                Precursor  -- I have added this explicit call to make the Eiffel code
    equivalent to the Java version, see previous example. [EP6]
                data := d.cloned
            end

        update do .... print (data.size) ... end
end

class SUBJECT feature
        register (o: OBSERVER) do ... o.update (...) end
end
```

This example is invalid with the above rules since:

- The precursor *make* (from *DEMO*) is *NEW_DEMO*-creation-involved since it is called by *NEW_DEMO*'s *make*.
- Routine *register* has *o* as a targeted formal argument (3.3.1).
- Hence **Current** is targeted at EP5 (3.3.6).
- Encure Current is targeted at EP6 (definition of ep').
- However, *data* is not properly set at EP5, violating 3.4.1.

If we reverse the order of the two instructions of the redefined *make*, the rule is no longer violated and the code becomes valid. (Somewhere in his slides Müller says that the order does not matter, but here it does.)

4.7 MS, "field read" example

I don't entirely understand this example as it only comes with a solution with the "committed" keyword and it is not in the paper. I am trying my best to figure out what is involved. I have changed the class name from Node to Node1 to avoid confusion with the paper's first example.

Original text:

```
class Node1 {
 Node! next;  // a cyclic list
 Object? elem;
 boolean contains( Object? e ) {
    committed Node! ptr = this.next;
    while( ptr != this ) {
     if( ptr.elem.equals( e ) )
       return true;
     ptr = ptr.next;            //ptr has to be committed
    }
   return false;
 }
}
```

Eiffel version, without the **committed** keyword:

```
class NODE1 feature
        next: NODE1
        elem: detachable ANY

        contains (e: detachable ANY): BOOLEAN
                local
                        ptr: NODE1
                do
                        from
                                ptr := next
                        until
                                ptr = Current or Result
                        loop
                                Result := (ptr = Current)
                                ptr := ptr.next
                        end
                end
end
```

If anyone understands the example better I would appreciate explanations.

4.8 MS, married persons

This example is alluded to in Müller's slides and I assume it's a reference to my Dependent Delegate Dilemma paper [2]

Assume a PERSON class as follows:

```
class PERSON create
        make_single
feature
        is_married: BOOLEAN
        spouse: detachable PERSON
        name: STRING

        make_single (n: STRING) do name := n end

invariant
        is_married implies spouse /= Void and then spouse.spouse = Current
end
```

The paper discusses how to write a set_spouse procedure, but here assume we want to include a creation procedure make_with_spouse:

```
make_with_spouse (n1, n2: STRING)
        do
                make_single (n1)
                create spouse.make_single (n2)
                spouse.set_spouse (Current)

        end
```

Again I would appreciate some explanations as to the purpose of this example

4.9 Simple example 2

I am not sure where this example comes from. I would appreciate if its author could clean it up (see in particular the line EP3) so that we can include it.

```
class A create
   make
feature
   make
      do
         create s
         s.do_something (Current)                    -- EP1
         s.call_something_on_previously_given_argument -- EP2
         s.return_previously_given_argument.s2.do_something -- EP3
         create s2
      end

   s, s2: S
end
```

```
class S feature
   do_something (a: A) do    value := a  end


   call_something_on_previously_given_argument
      do value.s2.do_something    -- May crash if `s2' is not set yet end
   return_previously_given_argument: A
      do    Result := value end
   value: A
end
```

5. Discussion

We have not attempted any formalization or proof.

The results would be hard to publish because the verification community is convinced that verification must be "modular".

In fact the mechanism may be modular enough for our purposes. For efficient implementation, the compiled version of a routine should record whether formals are targeted. This leaves the problem of a deferred routine, which according to this criterion is untargeted (this is sound thanks to rule 3.4. The only constraint for the programmer, then, is that if any descendant is going to include a call targeted to the formal, it should include a Precondition (or Postcondition)

f.available

This seems like a reasonable requirement – in any case, better than imposing a whole new language mechanism. Note that in the SCOOP context where preconditions are wait conditions this particular kind of precondition should not cause a problem, since the result is always true (and the precondition might conceivably be optimized away, especially if *available* is frozen).

A problem that does remain, however, is that of a detachable f, for which, as A. Kogtenkov has pointed out, the precondition should be written

attached *a* **implies** *a.available*

Unfortunately I don't see any better way in this case.

Another issue is to make sure a violation of the validity rules leads to clear error messages, so that programmers, especially novice ones, get a good idea of what is going on. But I

think it's an easier problem than explaining a new language mechanism that is only useful to handle very special cases.

As a final note (prompted again by AK), a creation procedure whose target is not targeted does not have to satisfy the invariant on exit. Both the Hoare-style theory and the run-time assertion monitoring mechanism should be updated accordingly.

References

[1] Bertrand Meyer, Alexander Kogtenkov and Emmanuel Stapf: *Avoid a Void: The Eradication of Null Dereferencing*, in *Reflections on the Work of C.A.R. Hoare*, eds. C. B. Jones, A.W. Roscoe and K.R. Wood, Springer-Verlag, 2010, pages 189-211, available at http://s.eiffel.com/void_safety_paper.

[2] Bertrand Meyer, *The Dependent Delegate Dilemma*, in *Engineering Theories of Software Intensive Systems*, Proceedings of the NATO Advanced Study Institute on Engineering Theories of Software Intensive Systems, Marktoberdorf, Germany, from 3 to 15 August 2004, eds. Manfred Broy, J Gruenbauer, David Harel, C.A.R. Hoare, NATO Science Series II: Mathematics, Physics and Chemistry, vol. 195, Springer-Verlag, June 2005, available at http://se.ethz.ch/~meyer/publications/lncs/ddd.pdf.

[3] Alexander Summers and Peter Müller, *Freedom Before Commitment Simple Flexible Initialisation for Non-Null Types*, in OOPSLA 2011, available at http://pm.inf.ethz.ch/publications/getpdf.php?bibname=Own&id=SummersMuellerTR11.pdf.

[4] Peter Müller, presentation slides for [3].

[5] EiffelVision library documentation, at http://docs.eiffel.com/book/solutions/eiffelvision-2.

[6] Ecma International: *Eiffel Analysis, Design and Programming Language*, approved as International Standard 367 by ECMA International, 21 June 2005; revised edition, December 2006, approved by the International Standards Organization as the ISO standard ISO/IEC 25436:2006.